Memoria Miniproyecto

Integrantes del grupo
Javier Martínez Llinares
Pedro de Luna Huerta
Lucia Carro Arcaya
Lidia Moreno Marín
Amparo Gálvez Vilar
Antonio Hernando Graboleda

Introducción

Este proyecto nace con la intención de relacionar al tráfico y la contaminación en Valencia. Para medir la contaminación utilizamos los datos de emisiones contaminantes recabados en diversas estaciones repartidas por la ciudad y para medir el tráfico usamos la Intensidad Media Diaria (IMD). Ya con estos datos buscaremos la posible relación que guarden entre sí. Además, una de las conclusiones que se busca extraer es ver cómo afecta la presencia de los parques urbanos a la contaminación y al tráfico. Si en zonas con parques urbanos la contaminación o el tráfico son elevados podremos afirmar que estos parques no se están cuidando como deberían. Con este proyecto se busca extraer conclusiones claras acerca de estos temas, con el objetivo de mostrar a la gente un problema patente que está pasando desapercibido.

Hemos valorado el añadir otros datasets como las bicicletas que circulan, pero lo hemos despreciado, ya que hemos considerado que no influyen en la contaminación.
Si hay bicicletas (probablemente), circulan menos coches, pero si ya tenemos la cantidad de coches que circulan las bicicletas no aportarán ninguna información respecto a la contaminación.

Sin embargo, sí que hemos incluido los datos de las zonas verdes. Esto lo hemos hecho para pintar los datos en los mapas, ya que existe la posibilidad de que la contaminación depende si hay zonas verdes o no. Otro punto es que la contaminación puede ser muy negativa y afectar a las zonas verdes. También es cierto que a la hora de representar el mapa de Valencia pintar el viejo cauce del Turia ayuda muchísimo a orientarse.

Datasets usados:
https://valencia.opendatasoft.com/explore/dataset/rvvcca/information/, de aquí hemos sacado los datos de la contaminación.
https://valencia.opendatasoft.com/explore/dataset/estacions-contaminacio-atmosferiques-estaciones-contaminacionatmosfericas/table/, de aquí hemos georeferenciado los datos anteriores.
https://valencia.opendatasoft.com/explore/dataset/qlik_datos_imd_mobilitat_enero_2016-diciembre2022_coord/information/, de aquí hemos sacado los datos del tráfico.
https://valencia.opendatasoft.com/explore/dataset/zonas-verdes/table/?disjunctive.nivel3, de aquí hemos sacado los datos de los parques urbanos.

Preparación de los datos

En la preparación de los datos vamos a importar los datos de la contaminación, vamos a eliminar aquellas columnas que tengan más de 10000 NA, para facilitar el análisis. Y vamos a añadir la referencia geográfica. Eliminaremos los datos de los cuales no tengamos referencia geográfica.

En el conjunto de datos de las espiras magnéticas para saber el tráfico hemos modificado la variable IDM para poder importarla desde qgis como variable numérica.

Finalmente hemos exportado los datos en csv, tanto de los datos de las espiras como de los datos de los georeferenciados, estos últimos lo haremos tanto íntegramente como separados por medida media, para realizar el procesado en qgis de manera adecuada.

Pre-procesamiento en qgis

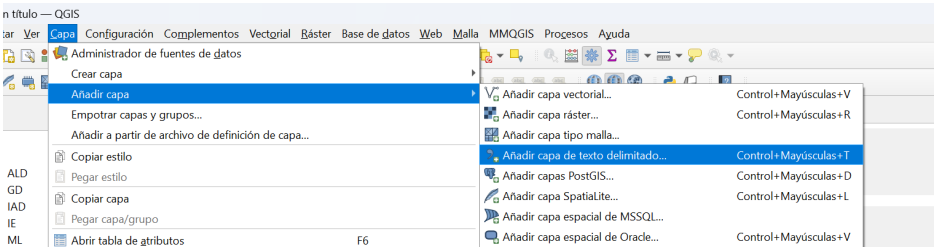
Vamos a hacer un pre-procesamiento en qgis, para ello vamos a hacer varios mapas para extraer los archivos que necesitamos.

Pre-procesado común

Dado que hay elementos comunes a ambos gráficos se han hecho una vez y se cogen desde ahí.

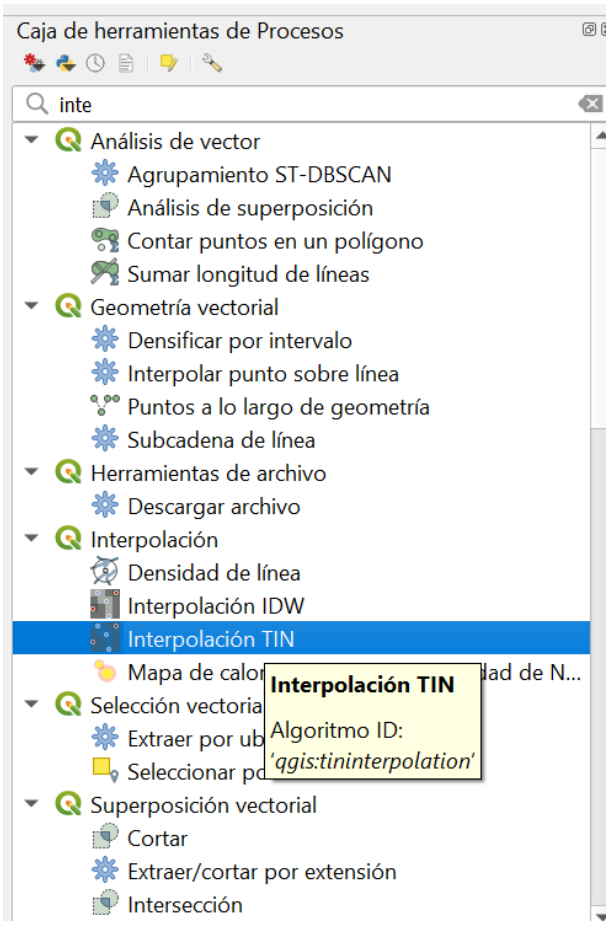
Interpolación

Añadimos una capa con los valores de imd.csv

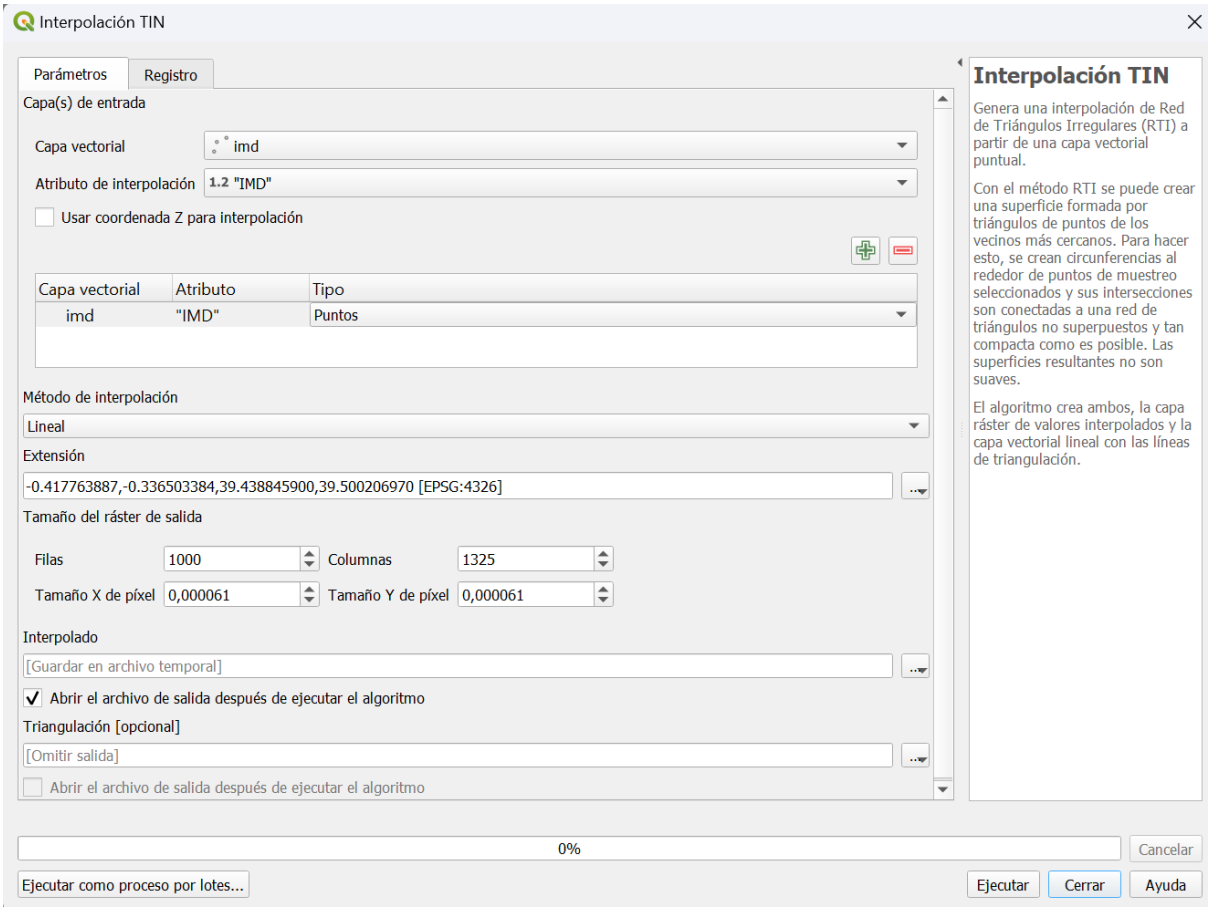


El resultado:

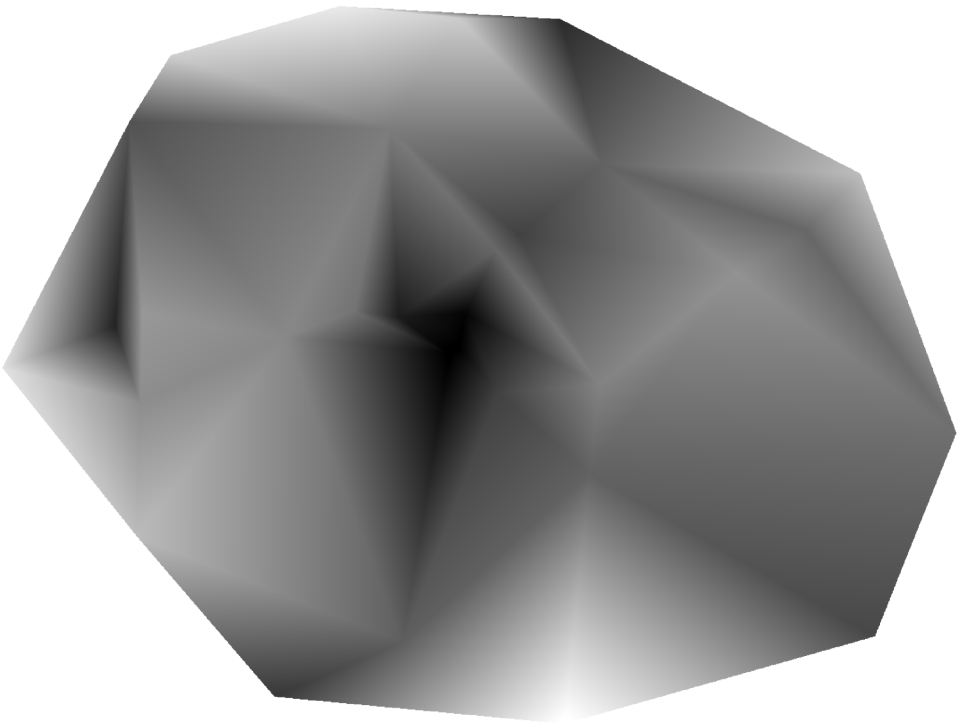
Buscamos la interpolación TIN en la caja de herramientas



Realizamos la interpolación del imd tal y como se muestra:

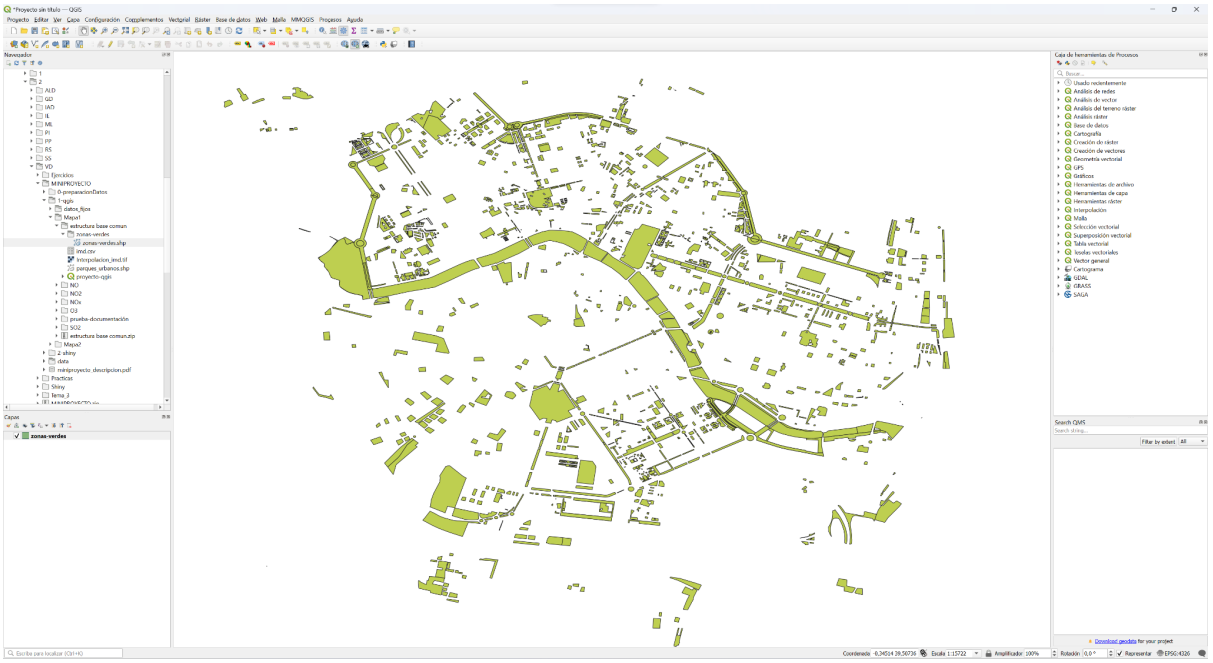


El resultado final es:



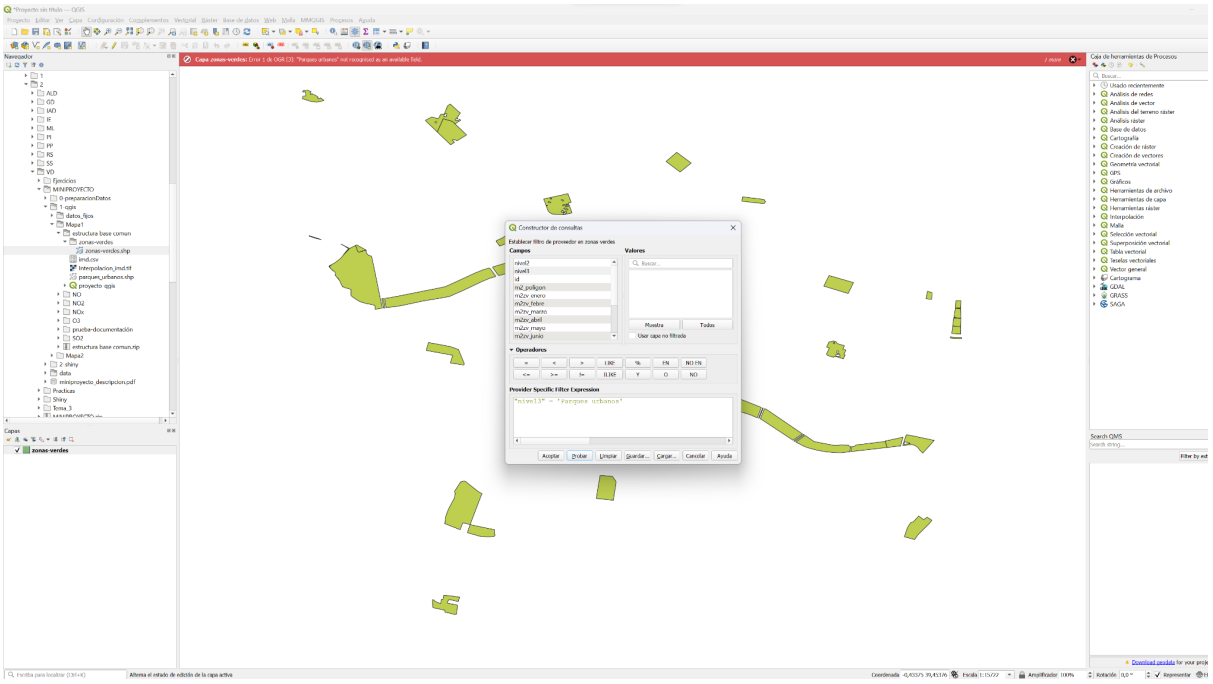
Parques

Primero cargamos las zonas verdes.



Como vemos son muchas, y hay algunas que son muy pequeñas, por lo que nos vamos a quedar únicamente con los parques urbanos, al ser menos y más grandes.

Para eso pulsamos sobre la capa el botón derecho y vamos a filtrar.



Una vez así exportamos la capa.

Mapa 1

De que se trata

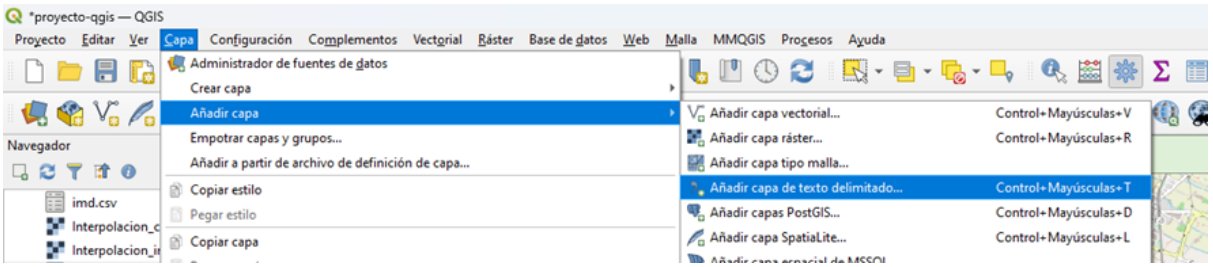
Este mapa muestra los datos de contaminación de un gas (NO, NO2, NOx, O3, SO2) recogidos en diversas estaciones. Esto lo realiza con los markers (que tienen un popup con el valor medio de dicha estación), y con los buffers, que de manera visual muestran un área afectada con un color en función de la cantidad de contaminación. Asimismo, el mapa muestra una interpolación del IMD (Intensidad Media Diaria), una medida de cuánto tráfico hay en diversas zonas. Finalmente, también se muestran los parques urbanos en verde. Este mapa nos permite ver si hay alguna relación entre la contaminación de los gases mencionados y la cantidad de tráfico. Además, fijándonos en los parques urbanos podemos ver si la cercanía a estos hace que haya menor contaminación o tráfico. De no ser así, quizás habría que replantearse si se están dañando estos parques urbanos, y buscar soluciones para reducir el tráfico y la contaminación en sus alrededores.

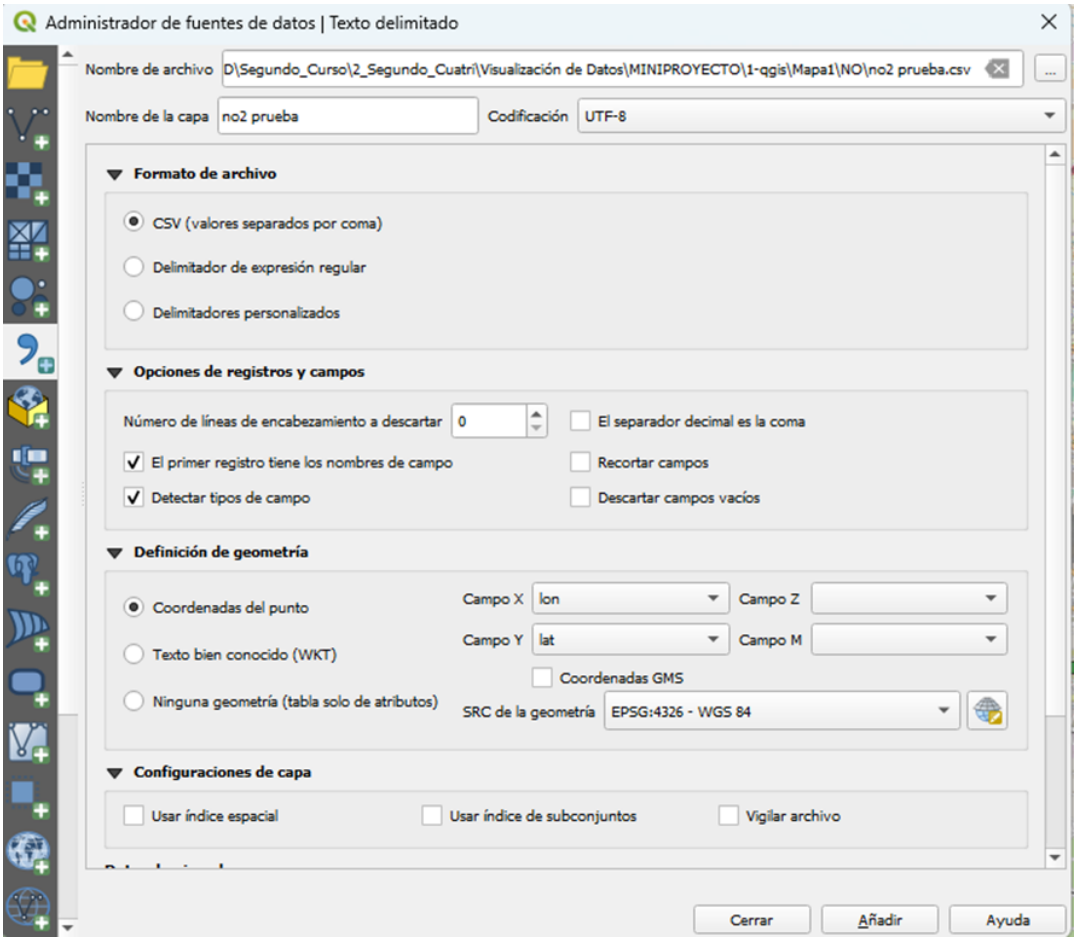
Cómo se hace

El proceso hay que hacerlo una vez para cada medida (NO,NO2,NOx,O3,SO2):

Consideraremos en el ejemplo la variable NO2

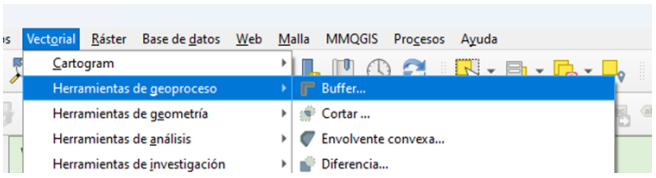
Añade una capa a partir del csv (creado con r con latitud, longitud y el valor medio de NO2 en cada estación)



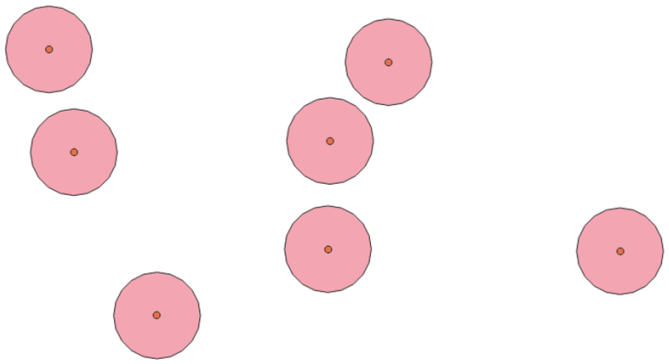


Salida:

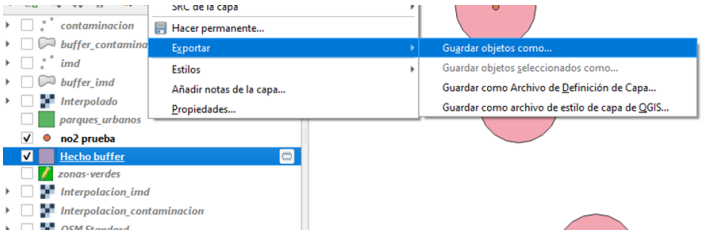
CREA UN BUFFER:

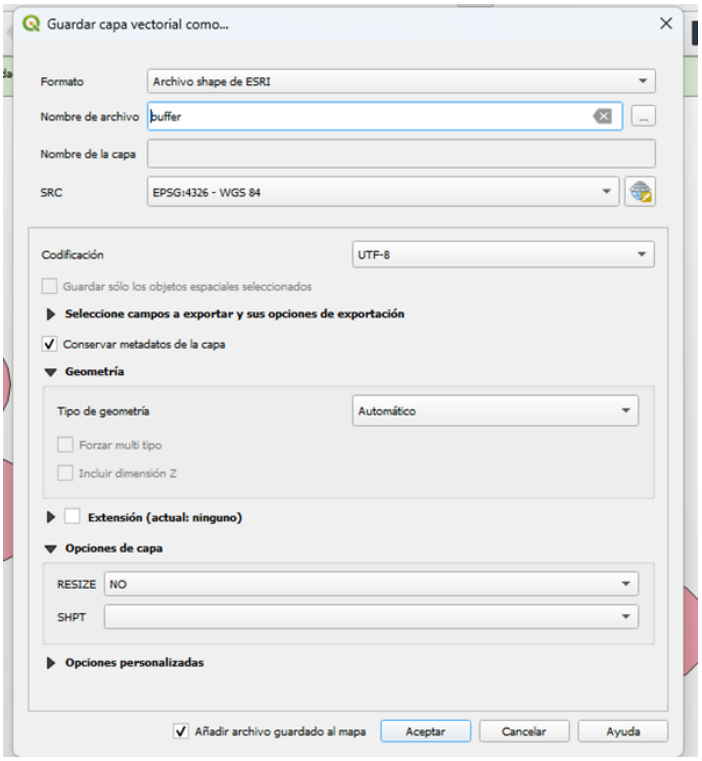


TE TIENE QUE SALIR ALGO ASI:



BOTÓN DERECHO SOBRE EL BUFFER CREADO:





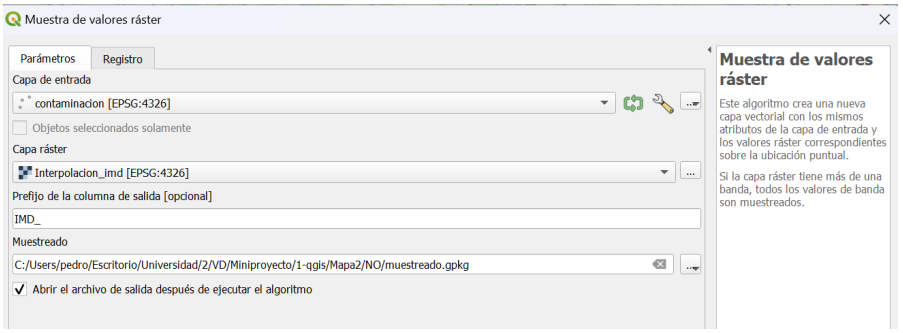
Mapa 2

De que se trata

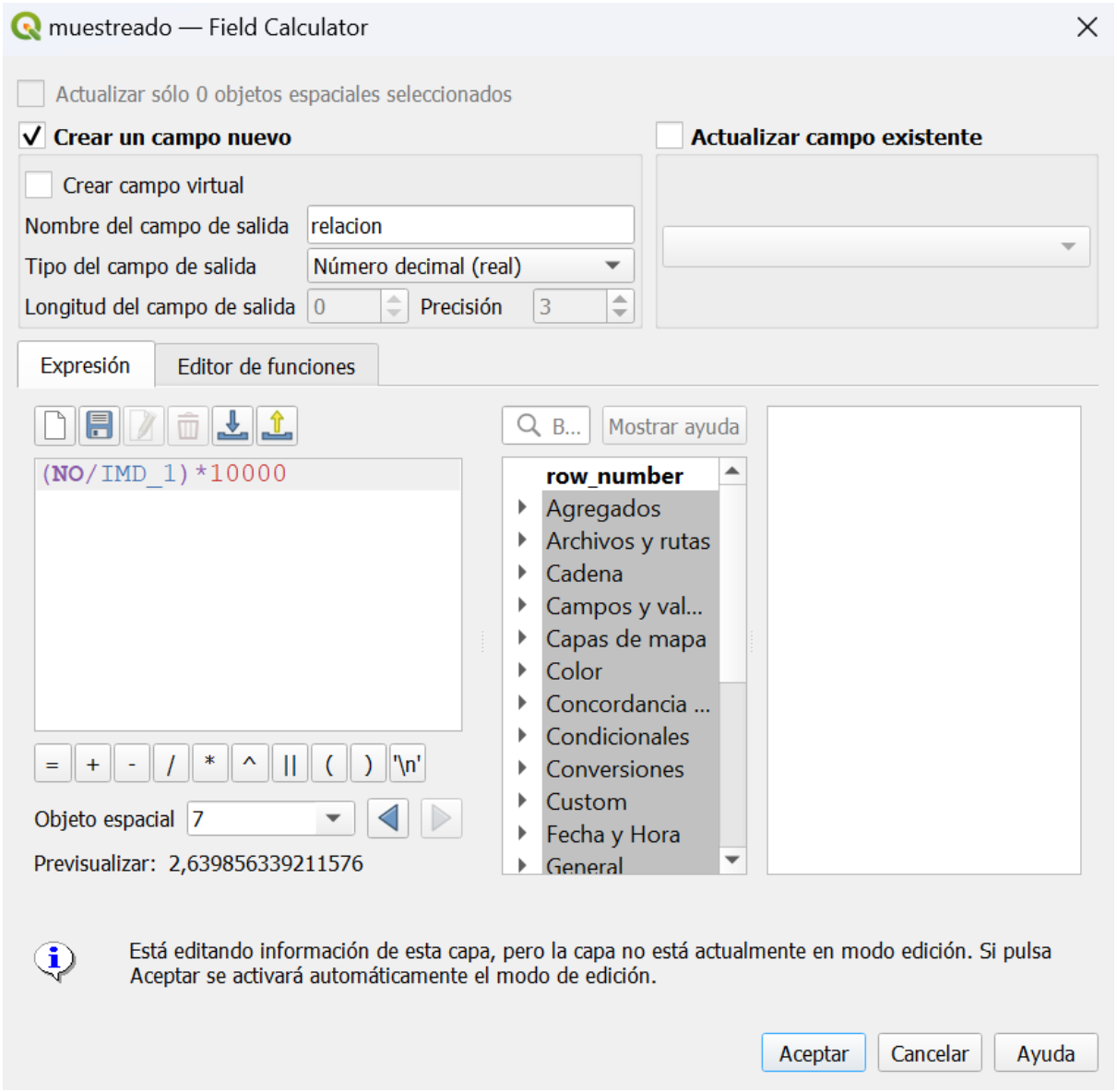
Este mapa muestra la relación entre el IMD y la contaminación de un gas (NO, NO2, NOx, O3, SO2). La relación está establecida como IMD/Contaminación, por lo que a mayor valor de la relación, es menor la cantidad de contaminación explicable por la cantidad de tráfico. Esto se representa con buffers situados en las estaciones, variando su color en función de cómo varía la relación. Esto nos permite ver si en zonas cercanas a los parques urbanos la contaminación se debe al tráfico o hay factores externos.

Cómo se hace

Abrir Muestra de valores de ráster.

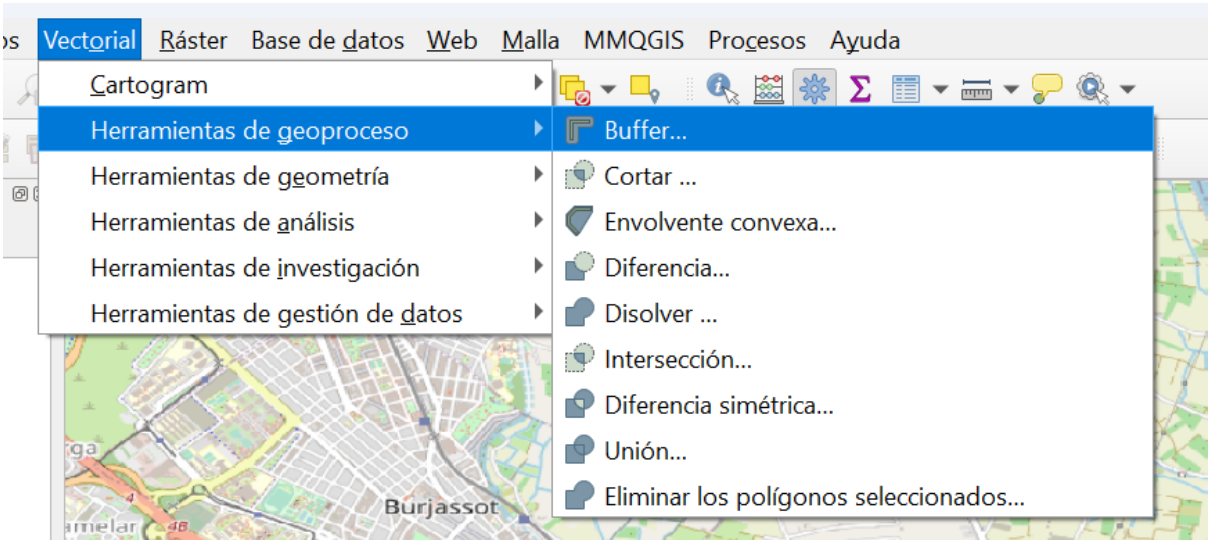


Seleccionamos la capa de contaminación(extraída del mapa 1), y la interpolación del IMD (extraída del mapa 1), y la guardamos en la carpeta del mapa 2 en (NO/NO2/NOx/O3/SO), en un archivo que se llame muestreado.
Abrir la calculadora de campos y poner esta configuración.

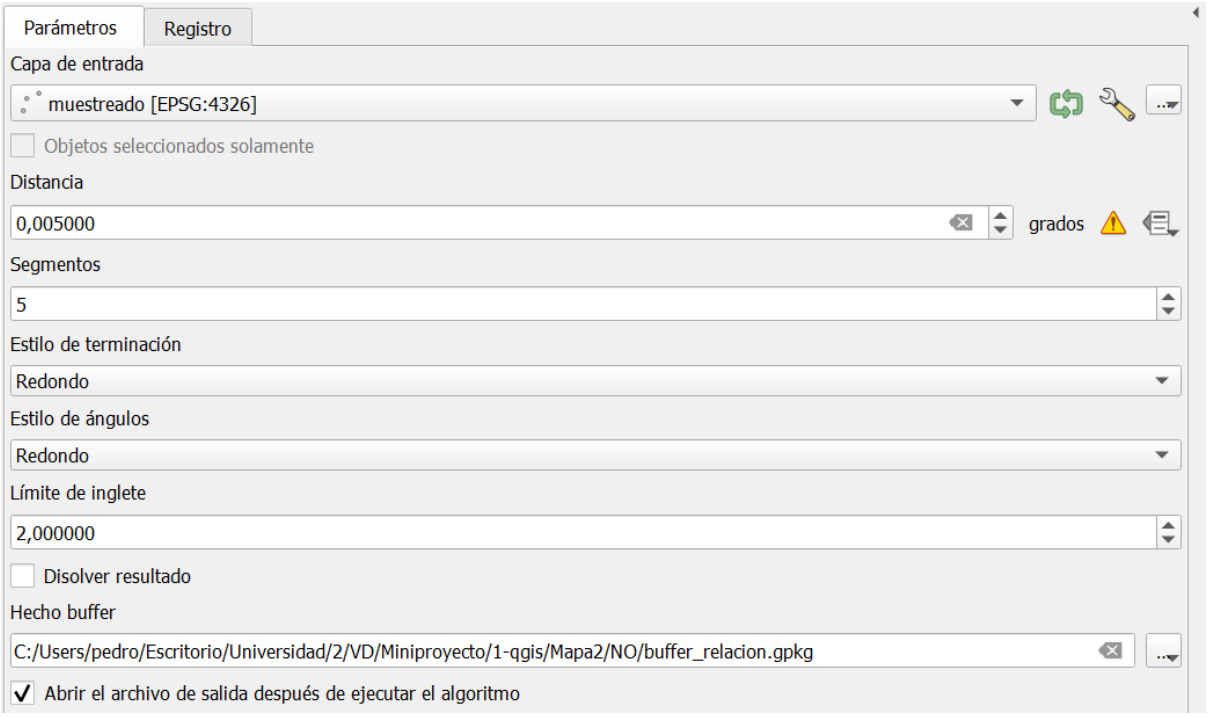


Fijarse en que hemos seleccionado la capa recién muestreada, y que el tipo es decimal.

Abrimos el buffer



Ponemos la siguiente info



Fijarse la capa de entrada es muestreado, la distancia es 0,005, y que hay que almacenar la capa en la carpeta del mapa 2 en un archivo llamado buffer_relacion.gpkg.

Finalmente exportamos las capas muestreado como csv y buffer_relacion como ESRI.

Shiny

Estructura general

Al inicio del trabajo hacemos la instalación (si es necesaria) y carga de las librerías necesarias.
Después realizamos la importación de los archivos necesarios. Como siempre son los mismos los hacemos fuera del server, ya que van a ser globales y nadie los va a modificar.
Tras estos pasos iniciales ya llegamos a la interfaz gráfica y al server.
Por la parte de la interfaz gráfica, se trata de un layout multi página, que está dividido en cuatro menús principales. El primero, muestra el desarrollo del trabajo, y se muestra este pdf y el video explicativo. El segundo menú es donde se hallan los mapas. En el tercero podemos encontrar los gráficos. Y en el cuarto es donde se puede ver quien ha desarrollado el trabajo.
Entrando en el server primero se han definido las llamadas que reciben de la parte del front, con sus respectivas salidas. Y después se han desarrollado las funciones (algunas reactivas y otras no), a las que llaman desde la primera parte del server.

Interfaz gráfica

Lo primero que llama la atención del código es que no se llama a la función fluidPage, sino a la dashboardPage, esta decisión la hemos tomado para usar los paquetes shinydashboard y shinydashboardPlus, lo que nos ha facilitado la organización del sidebar con los menús entre otras cosas.
Al inicio del código definimos la skin a “midnight”, que es como el tema pero de la dashboardPage.
Justo después utilizamos los tags para incluir un archivo css con algunas modificaciones que hemos hecho, así como un archivo javascript para añadir algunas funciones que aportan interacciones a nuestra web. Usamos la función tags$head para indicar que queremos crear tags en el módulo de html “head” (que es donde se suelen realizar las importaciones). Ahí usamos la función includeCSS para importar nuestro css y el tag$body para añadir un script que se abra con la ejecución del body (para que no se ejecute antes de esto dando error).

Primero definimos el sidebar, en el que para cada apartado asignamos un título, un id (al que haremos referencia en el body), y un icono. También se nos permite crear desplegables para introducir subtabs.
Posteriormente definimos el body. Aquí es donde está el contenido de la página. Definimos tantos tabItems como anteriormente tabNames hallamos puesto, cuando el tabName del elemento del menú coincida con el tabname del tabItem se mostrará el contenido de este en el body.

El primer tabItem es donde se hace referencia al pdf y al video, y donde esta el párrafo de introducción al proyecto. El video y el pdf estan en la carpeta www (la carpeta que toma como referencia shiny). El tab item tiene un fluidRow, que tiene dentro a su vez 2 columns (una de width 4 y la otra de 8). En la columna de 4 metemos con html puro un embed, con el cual mostramos el pdf. En la columna de 8 metemos también con html puto un texto y uta etiqueta <video> con la cual insertamos el video para que sea reproducible en nuestra app.

Estructura común de los gráficos y los mapas:
Esta estructura consta de un gráfico o mapa, que tendrá un ID, y unas dimensiones.
Un dropdownButton, en el que al pulsar podremos acceder a las modificaciones del gráfico, este botón está dentro de un div que hemos creado para poder poner un style = “” e introducir las coordenadas del mismo.
Este botón tiene, a parte de los inputs del gráfico las siguiente opciones:
right = TRUE, para que no se salga de la pantalla por la derecha.
inline = TRUE, para que el desplegable salga del botón.
up = TRUE, esto solo está en los gráficos, ya que el botón está abajo y así sale el desplegable por la parte de arriba y evitamos que salga de la pantalla.
status = "danger", definimos el color.
icon = icon("gear"), definimos el estilo del botón.
width = "300px", definimos el tamaño del botón.

A parte de esto también habrá en la parte inferior un texto explicativo de los gráficos/mapas.

En el dropdownButton del segundo tabItem (primer mapa) tiene:
Un texto con estilo h3.
Un selectInput que permite seleccionar el tipo de contaminación. Por defecto está marcado NO.
Un texto con estilo h5.
Tres switchInput, que son como un checkbox pero con estilo mejor de la librería shinyWidgets, el value=TRUE, define el estado inicial. Estos inputs sirven para mostrar o ocultar los buffers, la interpolación o los parques.
Después hay dos sliderInput, para controlar la opacidad del buffer y la interpolación.
Finalmente tenemos un actionButton, que hasta que no es pulsado no se actualizan los cambios, esto está definido en la parte del server.

El tercer tabItem no incluye nada que no tenga el segundo, sigue la misma estructura.

El cuarto tabItem hacer referencia al primer gráfico, y el dropdownButton tiene:
Un texto h3.

Un selectInput con los posibles tiempos por los que ver el gráfico. Por defecto por Años.
Otro selectInput con si se quiere ver agrupado por parámetros de contaminación o por estaciones. Por defecto están los parámetros.
Después hay un uiOutput(), esto llama a una función del server que en función del select input anterior muestra una cosa o otra. Si el select input son los parámetros nos muestra un checkboxGroup con los posibles parámetros, si es por estación nos muestra otro checkboxGroup pero con las posibles estaciones. Por defecto todas las opción están marcadas.
Finalmente, como este gráfico se podía modificar mucho hemos optado por no actualizarlo a cada modificación y hemos añadido un actionButton para que se invalide únicamente al pulsar.

El quinto tabItem hace referencia al segundo gráfico, y el dropdownButton tiene:
Un texto h3.
Un selectInput con los posibles tiempos por los que ver el gráfico. Por defecto por Años.
Un checkboxGroupInput, para poder seleccionar las áreas por las que filtrar, al haber cargado los datos previamente podemos cargar las choices sin necesidad de hacer un uiOuput().
Al no haber action button el gráfico se actualizará cada vez que se modifique un input.

En el sexto tabItem lo único especial es que tiene dos gráficos, estos gráficos los hemos metido en dos fluidRow, para que por muy ancha que sea la pantalla estén los dos siempre uno encima del otro para facilitar la comparación.

El último tabItem de gráficas no tiene dropdownButton, solo tiene una llamada a plotlyOutput.

El penúltimo tabItem es una apartado en el cual se muestra a los participantes del trabajo con colores y descripciones estilo startup. Esto se muestra usando la funcion userBox de shinydashboardPlus, a la cual le pasamos diversos parametros como la función los pide. Entre los que cabe destacar que la imagen está en la carpeta www.

El último tabItem es un sencillo apartado con un texto que narra las conclusiones del proyecto

Server

El server está muy organizado, tiene tres partes divididas.
En la primera hay una única función, que genera el uiOutput() del gráfico uno que hemos visto anteriormente.
En la segunda hay varias funciones con la siguiente estructura:

```
 output$Mapa1 <- renderLeaflet({
   return (funcionMapa1())
 })
```
Y en la tercera es donde están definidas las funciones funcionMapa1(), funcionMapa2(), funcionGrafico1() ...

Mapa 1

La función funcionMapa1(), es un eventReactive, que reacciona únicamente ante cambios de input$btn_m1, por lo que hasta que no se pulse al botón no se invalidará.

En primer lugar tomamos el valor del input del tipo de contaminacion y lo establecemos como nuestra medida. Después, con diversas herramientas de R extraemos el nombre de la variable que nos interesa de buffers[[medida]], ya que sabemos que el nombre de dicha variable contiene el nombre de la medida. Guardamos este nombre en VarBuffer y lo usamos para extraer de buffers[[medida]] los valores de dicha variable, que guardamos en valoresBuffer. Despues extraemos de markers[[medida]] la variable que contiene los valores de los puntos que usaremos para los markers.

A continuación creamos una paleta para los buffers personalizada (pal_buffer) con la funcion colorNumeric y los valores almacenados en valoresBuffer.

Ahora si creamos el mapa con la función leaflet, fijando con leafletOptions el zoom maximo y minimo. A continuación añadimos el mapa de teselas default y fijamos la vista con setView para que salga la ciudad de Valencia (coordenadas buscadas en internet).

Ya con el mapa añadimos los marcadores con la función addMarkers, pasandole los valores de longitud y latitud del df markers[[medida]] y el valor del popup de la variable creada anteriormente valoresMarkers. Además le pasamos como icono el icono exportado previamente del sensor “iconoSensor

Ahora comprobamos los inputs.

Si el de parques esta ON añadimos con la función addPolygons los parques urbanos (cuyos datos estan en la variable parques), pintandolos de color verde y totalmente opacos

Si el de interpolación esta ON añadimos con la función addRasterImage la interpolación (cuyos datos estan en la variable interpolacion),pintandolos según la paleta definida previamente “pal_inter” y fijando la opacidad en función del input de opacidad de interpolación. Añadimos también una leyenda con la función

addLegend abajo a la izquierda con la paleta pal_inter y los valores de la interpolación

Si el de buffers esta ON añadimos con la función addPolygons la interpolación (cuyos datos estan en buffers[[medida]]),pintandolos según la paleta definida previamente “pal_buffer” y fijando la opacidad en función del input de opacidad de buffers. Añadimos también una leyenda con la función addLegend abajo a la izquierda con la paleta pal_buffery los valores de buffers[[medida]].

Tras esto se devuelve el objeto mapa de leaflet que hemos creado.

Mapa 2

La función funcionMapa2(), es un eventReactive, que reacciona únicamente ante cambios de input$btn_m2, por lo que hasta que no se pulse al botón no se invalidará.

En primer lugar tomamos el valor del input del tipo de contaminacion y lo establecemos como nuestra medida. Despues, extraemos de buffers_relacion[[medida]]$relacion los valores de dicha variable, que guardamos en valoresBuffer.

A continuación creamos una paleta para los buffers personalizada (pal_buffer) con la funcion colorNumeric y los valores almacenados en valoresBuffer.

Ahora si creamos el mapa con la función leaflet, fijando con leafletOptions el zoom maximo y minimo. A continuación añadimos el mapa de teselas default y fijamos la vista con setView para que salga la ciudad de Valencia (coordenadas buscadas en internet).

Ahora comprobamos los inputs.

Si el de parques esta ON añadimos con la función addPolygons los parques urbanos (cuyos datos estan en la variable parques), pintandolos de color verde y totalmente opacos

Si el de buffers esta ON añadimos con la función addPolygons la interpolación (cuyos datos estan en buffers_relacion[[medida]]),pintandolos según la paleta definida previamente “pal_buffer” y fijando la opacidad en función del input de opacidad de buffers. Añadimos también una leyenda con la función addLegend abajo a la izquierda con la paleta pal_buffer y los valores de buffers_relacion[[medida]].

Tras esto se devuelve el objeto mapa de leaflet que hemos creado.

Gráfico 1

La función funcionGrafico1(), es un eventReactive, que reacciona únicamente ante cambios de input$btn_g2, por lo que hasta que no se pulse al botón no se invalidará.

Lo primero que vemos en la función es un condicional de: if(input$btn_g1 == 0), esto dice que si no se ha pulsado ninguna vez el botón lo que hay que hacer es por defecto definir los parámetros como toda la contaminación posible, esto es necesario porque por defecto si no se abre el dropdownButton sí que hay default de medida_g1 o de tiempo_g1, pero no de parametros_g1, esto es porque como no se ha llamado a input_g1 no se ha creado parametros_g1 y no hay valores por defecto, por eso hay que crearlo desde un inicio por defecto, una vez se le pulse una vez al botón el resto funcionará correctamente.

Hay 8 posibles combinaciones de gráfico, en función del tiempo y la categoría por la que se agrupe. Vamos a analizar dos gráficos (uno con el eje x contínuo y otro discreto), y el resto serán igual pero cambiando el parámetro de actualización.

Empezamos asignando a graf lo que va a salir y partiendo del df geo_data con la orden
graf = geo_data %>%
mutate(Fecha = ymd(Fecha)): esto transforma Fecha a tipo fecha con lubridate.
mutate(tiempo = year(Fecha)): esto coge únicamente el año, mes, día del mes, o de la semana. En función del gráfico que estemos haciendo.
gather(c(NO,NO2,SO2,O3,NOx),key = "param", value = "valor"): Esto sirve para reorganizar los datos mejor para que el ggplot funcione correctamente.
filter(is_in(param,parametros))/filter(is_in(Estacion,input$estaciones_g1)): Aquí aplicamos el filtro de checkboxGroupInput.
group_by(tiempo, Estacion)/group_by(tiempo, param), aquí agrupamos en función de la categoría por la que queramos agrupar.
drop_na(): eliminamos los na.
summarise(media = mean(valor)), calculamos la media para cada valor por tiempo y medida.
ggplot(aes(x = tiempo, y = media, colour = param)): define los parámetros del gráfico.
geom_line(): añadimos una capa.
labs(x = "", y = "", title = ""): aquí definimos los títulos de los ejes y del gráfico.
Dentro de theme(), definiremos los parámetros estéticos.
plot.background = element_rect(fill = "#353C42"): El fondo del gráfico sea del mismo color que el shiny.

panel.background = element_rect(fill = "#353C42"): Lo mismo de antes pero con el fondo del panel (donde no está el gráfico).
legend.background = element_rect(fill = "#353C42"): Lo mismo pero con la leyenda del gráfico.
plot.title = element_text(colour = "white",hjust = 0.5, size = 20): Definimos color, posición y tamaño del título del gráfico.
text = element_text(colour = "white"): Hacemos que el color de todo el texto sea blanco.
axis.line = element_line(colour = "white",size = 2): Hacemos las líneas de los ejes más gordas y de color blanco.
panel.grid.major = element_line(color = "lightgrey", size = 0.5): Hacemos las divisiones mayores del gráfico de color gris y de un tamaño menor.
panel.grid.minor = element_line(color = "lightgrey", size = 0.5): Hacemos las divisiones menores del gráfico de color gris y de un tamaño menor.
axis.text = element_text(colour = "white"): Definimos el color del texto del gráfico.
Ya fuera del theme:
scale_x_continuous(n.breaks = 10): para establecer el número de ticks en el eje x.
scale_color_brewer(palette="Set1"): para establecer los colores.

Ahora vamos a ver las modificaciones que se harían si el eje x fuera contínuo:
mutate(tiempo = factor(tiempo)): después de definir tiempo lo tendríamos que hacer factor.
ggplot(aes(x = tiempo, y = media, colour = param, group = param)): Deberíamos definir también los grupos del gráfico.
Y no haría falta la orden scale_x_continuous(n.breaks = 10).

Gráfico 2

La función funcionGrafico2(), es un eventReactive, que reacciona únicamente ante cambios de input$tiempo_g2 e input$area_coches_g2.

Este gráfico sigue el estilo del punto anterior, es función del tiempo hace uno u otro gráfico. La única diferencia es el datase, las variables y que al hacer el factor de el mes hemos tenido que especificar manualmente los niveles, para establecer el orden correctamente.

Gráfico 3

El gráfico 3 en realidad son dos gráficos, por eso hay dos llamadas a dos funciones distintas. Ambas son partes sacadas del gráfico uno y gráfico dos para poder ver la relación entre ellas. Se modifica el gráfico uno cuando hablamos de años para que cojan el mismo periodo. También se modifica el gráfico uno en el mes para hacer que el eje x tenga el mismo formato en ambas representaciones.

Gráfico 4

El gráfico 4 en realidad se trata de una función normal, esto es porque no se actualiza nunca, ya que no es necesario porque no tiene inputs.
Este gráfico de añadido tiene geom_smooth(method = "lm",se = T), que te añade una capa con una regresión lineal y te añade las bandas de error.
Y facet_wrap(.~param, scales = "free", ncol = 2), que sirve para hacer un gráfico por parámetro.

Anexo de extras

Clases css

Un css sirve para fijar los aspectos visuales (estilos) de una web. Esto se hace añadiendo propiedades como color o fuente de letra a los elementos que cumplen una determinada condición (como por ejemplo ser un div, tener un id concreto, tener una clase concreta....).

En el que hemos hecho nosotros hemos empezado importando la misma fuente que para el proyecto de shiny, esto es porque aunque en el proyecto ya está, si modificamos los estilos con css hemos de importar la fuente a parte.
La siguiente bloque es donde definimos que la fuente que queremos usar es la importada, y definimos donde la queremos usar, en este caso la queremos usar en todas esas etiquetas, a las etiquetas con los id que hay después del # , y a todas las que tengan selectize.input. Sin embargo no queremos usarla en aquellas que sean del tipo a y que tengan la clase “sidebar-toggle”, esto es porque en esos casos hacemos referencia a font-awesome, y si cambiamos la fuente el html lo interpreta mal y no pone el icono que queremos.

El próximo bloque hace que aquello que tenga el id de memoria tiene que ocupar el 90% de la altura total de la pantalla, y el 100% de la anchura total del div.

El penúltimo define que la anchura de aquello con id video será del 100% del div.

Finalmente fijamos el color a blanco de cualquier estructura p (párrafo) que esté dentro de una estructura em (cursiva). Esto se aplica en el apartado Nosotros a las descripciones de los integrantes del trabajo.

Funciones JS:

Hemos metido javascript en shiny para aplicar dos pequeñas funcionalidades que no se pueden lograr con shiny. El javascript no interactúa con R, sino que directamente interactúa con el html que generamos.

La primera función que tenemos sirve para que cuando minimizamos la barra lateral el nombre del proyecto desaparezca, y nos quedemos solo con el icono de R.

En primer lugar comprobamos que existe el body, ya que si no existiese la función daría errores. Si no existe muestra un mensaje por consola. Si existe ejecuta la función.

```
var body = document.querySelector('body');
if (body) {···
} else {
  console.log("No se pudo encontrar el elemento del cuerpo (body).");
}
```

En primer lugar creamos la función onSidebarCollapse. Esta función busca el span del html que contiene el logo (este span tiene la clase logo por eso .logo) y cambia su contenido a una imagen de fontAwesome (el logo de R).

```
function onSidebarCollapse() {
  var spanElement = document.querySelector('.logo');
  spanElement.innerHTML = '<i class="fa-brands fa-r-project"></i>';
}
```

En segundo lugar creamos las funciones onSidebarExpand. Esta función busca el span del html que contiene el logo (este span tiene la clase logo por eso .logo) y cambia su contenido a una imagen de fontAwesome (el logo de R) seguida de la palabra MiniProyecto

```
function onSidebarExpand() {
  var spanElement = document.querySelector('.logo');
  spanElement.innerHTML = '<i class="fa-brands fa-r-project"></i> MiniProyecto';
}
```

Después creamos un mutationObserver . Esto sirve para ejecutar una función cuando hay un cambio en el código html. Al mutationObserver le aplicamos una función que comprueba si la mutación se ha realizado en un atributo class,. Si es así comprueba si entre las clases del elemento mutado se halla la clase sidebar-collapse. Si es así ejecuta la funcion onSidebarCollapse, sino ejecuta onSidebarExpand.

```
var observer = new MutationObserver(function(mutations) {
  mutations.forEach(function(mutation) {
    if (mutation.attributeName === 'class') {
      var classList = mutation.target.classList;
      if (classList.contains('sidebar-collapse')) {
        onSidebarCollapse();
      } else {
        onSidebarExpand();
      }
    }
  });
});
```

Finalmente fijamos la configuración del observer y lo ejecutamos para que observe lo que hay dentro del body.

```
var config = { attributes: true };
observer.observe(body, config);
```

La segunda función que tenemos sirve para que el logo del item del menu que este activo tenga un pequeño efecto de movimiento (facilitando ver en que item del menú estamos)

En primer lugar creamos la función controlarDivs.
Adquirimos todos los divs cuyo id empiece por shiny-tab (son los divs correspondientes a cada item del menú.)

```
var divs = document.querySelectorAll('div[id^="shiny-tab"]');
```

Para cada uno de ellos ejecutamos el siguente código:

```
divs.forEach(function(div) {
  if (div.classList.contains('active')) {
    var divId = div.id;
    var activeLink = document.querySelector('a[href="#' + divId + '"]');
    var menuIcons = document.querySelectorAll('.sidebar-menu i');


    menuIcons.forEach(function(icon) {
      if (icon.parentElement.href.split('#')[1] == activeLink.href.split('#')[1]) {
        icon.classList.add('fa-beat');
        if(icon.parentElement.parentElement.parentElement.classList.contains("treeview-menu")){
            icon.parentElement.parentElement.parentElement.parentElement.firstElementChild.firstElementChild.classList.add('fa-beat');
        }

      } else {
        icon.classList.remove('fa-beat');
      }
    });
  }
});
```

En primer lugar, comprobamos si el div tiene la clase activo, si es así guardamos su id en una variable. Seleccionamos los links cuyo href hace referencia al div activo. Además seleccionamos todos los i que hay en el sidebar-menu (son los logos de cada ítem y subitem).

Para cada i comprobamos si el href al que hace referencia es igual que el href al que hace referencia el link activo (si son iguales). Si no lo son eliminamos la clase fa-beat (que es una clase de fontAwesome que aporta movimiento a los logos). SI si que lo es le añadimos la clase fa-beat. Además también comprobamos si está dentro de un treeview-menu (es decir, si es un sub item), y si es así también le aplicamos la clase a el logo del ítem principal.

En segundo lugar ejecutamos la función beatingLogo. Lo primero que hace esta función es comprobar que el contenedor con clase tab-content existe. SI existe crea un MutationObserver que para cada mutación que llegue ejecuta esta condición:

```
if (mutation.type === 'attributes' && mutation.attributeName === 'class' && mutation.target.classList.contains('active'))
```

Esta condición indica si lo que ha variado es un atributo, si es concretamente el atributo clase y si el elemento mutado contiene la clase active. Si esta condicion se cumple ejecuta la función controlarDivs.

Posteriormente fijo las opciones del observer y lo ejecuto:

```
var opciones = {
  attributes: true, // Observar cambios en
  attributeFilter: ['class'], // Observar
  subtree: true // Observar cambios en los
};

// Comenzar a observar el contenedor
observer.observe(contenedor, opciones);
```

Finalmente, ya fuera de la función beatingLogo inserto un eventListener que hará que al cargarse la página por primera vez se ejecuten las funciones controlarDivs y beatingLogo.

```
window.addEventListener('load', controlarDivs);
window.addEventListener('load', beatingLogo);
```

Resultados

Como se puede ver en el mapa 1 y en el gráfico 4 parece no haber mucha relación entre la cantidad de coches y la contaminación.
De hecho en el gráfico 4 podemos ver como la pendiente del modelo de regresión es prácticamente 0, lo que dice que no existe relación.
Sin embargo, si observamos el mapa 2 se ve que hay cierta relación. Buscando por internet hemos encontrado que los coches (principalmente) emiten nitratos, por eso vamos a centrarnos más en ese tipo de contaminación. En el mapa dos, si seleccionamos los nitratos (NO,NO2,NOx) se puede ver que en el centro de la ciudad la mayor parte de la contaminación no es explicada por los coches (contaminación/IMD es alto), pero en las afueras donde no hay tantas fuentes emisoras de contaminación la relación es más baja, porque la mayoría de la contaminación viene dada por el tráfico.
En el gráfico 3 se confirma lo que hemos podido ver en los dos anteriores. Que existe una línea de tendencia más o menos común. Los mínimos y los máximos se hallan próximos, en ambos gráficos.

Conclusiones

A pesar de que la lógica nos dice que a mayor cantidad de coches sí o sí debería haber más contaminación con los datos que hemos obtenido no podemos afirmar tajantemente que haya una relación. Es muy probable que exista dicha relación, pero con los datos que nos ha facilitado la Generalitat no podemos afirmarlo.
A pesar de ello, sí que podemos decir que la contaminación es un fenómeno multifactorial y que no depende únicamente de los coches.
También hemos visto como las medidas puntuales carecen bastante de sentido. Por ejemplo, si por esta calle no pasa ningún coche pero por la de al lado no paran de pasar la contaminación en esta calle será más alta de lo que debería. Dado que los contaminantes se mueven por el aire tiene bastante sentido hacer la comparación por tiempo y no por localización, como se ha visto en el mapa tres, donde se ha visto que existe una mayor relación.